Battlemesh Test Documentation Documentation

Release 1

battlemesh.org

Nov 03, 2017

Contents

This repository contains the documentation and results of tests executed during the Wireless Battle of the Mesh (starting from the v8 edition).

Editions:

CHAPTER 1

Battlemesh v9, Portugal (2016)

Battlemesh v9, Porto (Portugal) on 1st-7th May 2016.

Contents:

CHAPTER 2

Battlemesh v8, Slovenia (2015)

Battlemesh v8, Maribor (Slovenia) on 3rd-9th August 2015.

Contents:

2.1 1. The Mesh of Death Adversity

2.1.1 Topology and scenario

(Amadeus Alfa, Juliusz Chroboczek, Federico Capoano, Goran Mahovlić, Tomislava, Dario Stelitano, Riccardo Bloise)

Our test network consists of 10 wifi routers all directly connected to each others with one exception: *A* and *K* are not directly connected. Each router operates simultaneously with two different frequencies, one in 2.4 GHz, and the other in 5 Ghz. To avoid a link between *A* and *K*, we put them in different rooms and diminished their transmission power, while other routers (*B* to *J*) were in a large hall.

In order to test the performances of this network, two hosts are connected with wires at the extremities of the network: the client to *A*, and the server to *K*. We use wires to avoid interfering with the network we were testing. Then, the client performs some tests, using two tools: `ping`, to measure the latency between the two hosts, and `iperf`, to stress out the network and measure the throughput.

In this scenario, we are comparing 5 actively maintained and deployed routing protocols: Babel, Batman-adv, BMX7, OLSRv1 and OLSRv2.

Note:

- Some links are not drawn to avoid unnecessary confusion,
- Remark that the shortest path from *A* to *K* consists of 2 hops,
- There is no router *I* to avoid a possible confusion with the number 1.

2.1.2 Problems

This topology looks like really simple, but we expect that the interferences generated by as many routers should be sufficient to make differences between protocols.

Indeed, as described in The Hidden Node Problem, a wifi adapter is either transmitting or receiving. From a routing point of view, this means that doing two hops on the same channel will divide the throughput by two, and with three hops by three, etc. Using multiple non-interfering channels can avoid this problem, or at least will limit it: the transit node of a two hop route with each hop on a different channel can receive on one channel packets that it simultaneously send on the other channel. Another problem with wifi is that the performances decrease quickly with the distance because of packet loss: it is often better to take a two hop route instead of a long one hop route, even if the two hops are on the same channel.

From a routing point of view, we see that there is a real challenge: choosing the right tradeoff between taking a few hops, dodging lossy links and varying the channels used.

2.1.3 Requirements

- 10x Tp Link WDR4300 with OpenWRT
- 2x laptops with real ethernet ports (no adapters)
- 2x ethernet cables

2.1.4 Configuration

Note: All the configuration files for each router are available on github.

The binary of the firmware is also available.

Each node is a dual radio wireless router (TP-Link WDR4300), the most important facts related to the configuration are:

- multi channel mesh (2 GHz and 5 GHz)
- dual stack (IPv4 and IPv6)
- protocols installed: **Babel**, **Batman-adv**, **BMX7**, **OLSRv1** and **OLSRv2**
- laptops were connected to the mesh with static routes on nodes *A* and *K*

Warning: By the end of the eight edition we came to the conclusion that having to set up static routes to plug laptops into the mesh was a mistake.

We also haven't been able to run batman-adv with the same network configuration of the other routing protocols.

For these reasons Henning Rogge proposed a better configuration plan for the next edition (Battlemesh v9).

2.1.5 Test

(Henning Rogge, Thijs Van Veen)

Note: The test script is available on github, the relevant sections are test 1, 2 and 3.It has been carefully crafted such that the tests can be repeated easily.

The tests mainly consisted in generating traffic from the client connected to *A* to the server connected to *K*. The measurements were collected on the client.

3 different tests were performed:

- **reboot**: measure ping RTT while the mesh is rebooted
- **ping**: only measure ping RTT
- **ping + iperf**: measure ping RTT and throughput of a 10 Mbit/s UDP Iperf stream running simultaneously

Note:

- **RTT** stands for Round Trip Time
- **ECDF** stands for *Empirical Cumulative Distribution Function*

2.1.6 Results

(Matthieu Boutier)

Graphs and raw data are provided for each test.

Note: The graphs were generated with the generate_graphs.sh script (requires the R programming language), available on github.

reboot

In the *reboot* experiment, we let the network run stable for some time, and then suddently reboot all routers simultaneously. The following graph show a quick overview of the whole experiment.

(**How to read:** lower is better) What interests us in this experiment is the small part after the reboot: the following graph represent the *ECDF* graph of the ping samples taken for 50s after the reboot. The x-axis is scaled to show only packets than less than 50ms: we see that all protocols are choosing fast routes, since in all cases, the RTT of the packets are below 50ms. In this particular example though, Babel, BMX7 and OLSRv1, with almost all packets being under 10ms, outperforms Batman-adv and OLSRv2, which "only" have 80% of the packets under 10ms.

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Zooming at the normal graphs around time 150 gives us another precious informations: we see when the routing protocols begin to forward packets, which should reflect the convergence time of each protocol. Regarding this benchmark, we observe the following convergence time:

Babel	OLSRv2	BMX7	OLSRv1	Batman-adv
151	155	159	163	182
+0	+4s	+8s	+12s	+23s

(**How to read:** lower is better)

Note: Raw data for this test is available on github.

ping

In the *ping* experiment, we just measure the latency of the network with the *ping* tool, without any other perturbation. We expect an extremely stable network, with low RTT measurements and high fairness.

The following graph shows that for all protocols except Batman-adv, packets are routed pretty fairly, and have for 90% of them less than 5ms RTT, and for almost all of them less than 10ms. Packets routed by Batman-adv are not routed fairly: 50% are less than 4ms, 80% are less than 8ms, 90% are less than 10ms and almost all are less than 50ms.

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Looking the details shows that OLSRv1 and BMX7 are leading to the fairest and fastest RTT, behaving exceptionnaly well. They are closely followed by Babel, which has a slight fairness pathology: most of the packets (80%) are around 3.2ms, but around 10% are around 4ms, with a visible irregularity. Then comes OLSRv2, very fair, with packets around 4.5ms (+1ms).

The Babel irregularity can be explained with the following graph, giving only the Babel curve. We see that the packets having a higher RTT value are grouped in two points. This may happen because Babel hesitate with two paths, and sometimes switch to the wrong one: he then takes around 15s to decide that the other route was better, and stay much longer (70s minimum) on that better route.

Measured RTT in classic graph (Babel only):

(**How to read:** lower is better)

Note: Raw data for this test is available on github.

---

ping + iperf

In the *ping + iperf* experiment, we measure the latency of the network with the *ping* tool while pushing 10MB/s additionnal traffic from the client to the server. The graph below shows that OLSRv2 gives the fairest results for all packets: and has 95% of its packets are under 20ms, against 40ms for Babel and BMX7, 80ms for OLSRv1, and around 820ms for Batman-adv.

Measured RTT in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Interestingly enough, for 75% of the packets, Babel is leading with RTT under 9ms, but doesn't loose its fairness like the previous test, with a visible step: it's merely progressive. OLSRv2, BMX7 and OLSRv1 gives RTT under 13ms, and Batman-adv under 65ms.

Measured RTT in classic graph:

(**How to read:** lower is better) Finally, all protocols lead to the expected bitrate (10MB/s), as we see on the following graph.

Measured Bitrate:

(**How to read:** higher is better)

Note: Raw data for this test is available on github.

Article written by Federico Capoano, Matthieu Boutier, Thijs van Veen.

2.2 2. The Crossed Streams Jeopardy

Warning: To perform the crossed streams tests we had to reuse the topology prepared for *The Mesh of Death Adversity Test* because we were running out of time.

Therefore this scenario inherits all the settings of the previous test but adds an additional stream of traffic to stress test the network.

2.2.1 Topology

Same as *The Mesh of Death Adversity Test* with a small variation:

- there is a second stream of iperf traffic from node `D` to `H`

2.2.2 Test

Note: The test scripts are available on github:

- 10mbit iperf streams
- 100mbit iperf streams

The tests mainly consisted in measuring ping *Round Trip Time* and throughput from **client** to **server** while generating two streams of traffic:

1. from the **client** connected to **A** to the **server** connected **K**
2. from node **D** (*cross client*) to node **H** (*cross server*)

The measurements where collected from **A**.

2 different tests were performed:

- **10mbit iperf streams**: measure ping RTT while 2 simultaneous 10 Mbit/s UDP Iperf streams cross the network
- **100mbit iperf streams**: measure ping RTT while 2 simultaneous 100 Mbit/s UDP Iperf streams cross the network

Note:

- **RTT** stands for Round Trip Time
- **ECDF** stands for *Empirical Cumulative Distribution Function*

2.2.3 Results

Graphs and raw data are provided for each test.

Note: The graphs were generated with the following command (requires the R programming language):

```
R --vanilla --slave --args --out-type svg --separate-output --maxtime 300 --maxrtt␣
↪500 --width 9 --height 5.96 --palette "#FF0000 #005500 #0000FF #000000" results/ <␣
↪generic.R
```

the script generic.R is available on github.

10mbit iperf streams

Measured RTT in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Measured RTT in classic graph:

(**How to read:** lower is better) Measured Bitrate (from **client** to **server**):

(**How to read:** higher is better)

Note: Raw data for this test is available on github.

100mbit iperf streams

Measured RTT in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Measured RTT in classic graph:

(**How to read:** lower is better) Measured Bitrate (from **client** to **server**):

(**How to read:** higher is better)

Note: Raw data for this test is available on github.

Article written by Federico Capoano, Matthieu Boutier.

2.3 3: Blowing up the network

Warning: To perform this test we had to reuse the topology prepared for *The Mesh of Death Adversity Test* because we were running out of time.

Therefore this scenario inherits all the settings of the previous test but adds parallel streams of traffic between **A** and **K** in order to deliberately cause disruption in the network.

2.3.1 Test

Note: The tests were performed with Flent (FLExible Network Tester), the shell script that launches the 5 different tests is available on github.

The tests mainly consisted in generating a high amount of traffic between **client** and **server** while measuring network performance.

We ran five different tests on the test setup:

- Realtime Response Under Load (RRUL) test
- Realtime Response Under Load Best Effort (RRUL_BE) test
- TCP download test
- TCP upload test
- 8-streams download test, designed to mimic the dslreports speedtest

The point of this test series is to see what happens when the network is loaded to capacity (and beyond). Hence the "Blowing up the network" title of this test series. Since we only did a single test run, drawing conclusions from comparisons is not possible; but the tests can give an indication of the kind of behaviour that occurs at high load and point out areas for further investigation. And more rigorous testing with, above all, more repetitions can be performed to actually draw conclusions.

Realtime Response Under Load (RRUL) tests

The **RRUL** test consists of running four simultaneous bulk TCP streams in both the upstream and downstream directions (so eight streams total), while simultaneously running UDP and ICMP latency measurements. The test runs for 60 seconds, to make sure the network is fully saturated.

Two variants of the test were performed:

- the straight **RRUL** test, which marks each of the four TCP streams with different diffserv markings matching the mapping into hardware queues in the WiFi stack
- best-effort only (called **RRUL_BE**) in which no diffserv marking is employed

A timeseries graph of the behaviour of the RRUL test looks like this: The TCP streams start up five seconds after the ping measurements start, which is the flat part of the bottom-most graph. Then, after the TCP flows start, the latency goes up significantly, peaking at several seconds of latency and quite a bit of packet loss (seen as gaps in the graph). The throughput figures are likewise erratic and throughput is highly asymmetric. Finally, the UDP latency measurement flows are lost entirely; this happens because Netperf stops the measurement flows when encountering packet loss.

The name of the data set the above graph came from is deliberately omitted. The point here is not to beat up on a particular protocol, but to show the general pattern of failure experienced. As noted above, one should be wary comparing the different protocols from this data set, since there was only one test run. Instead, consider this an indication that they all break down, and that further investigation (and fixes!) is needed.

Aggregate results from all five protocols is available below, along with a link to the dataset which can be explored in further detail with the Flent tool, which can also run the tests for those wishing to explore the behaviour of their own network.

TCP traffic tests

The TCP traffic tests are just a single TCP stream with simultaneous latency measurement.

Two different tests were performed:

- TCP download
- TCP upload

8-streams download test

The **8-stream** download test runs eight simultaneous download streams while also measuring latency.

This test is designed to mimic the dslreports speedtest.

2.3.2 Results

Graphs are provided for each test.

Raw data is available on github.

Note: The graphs were generated with Flent (FLExible Network Tester) from the raw data collected in flent data files.

Realtime Response Under Load (RRUL)

ICMP latency in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Download bandwidth, upload bandwidth and ICMP latency in box graph:

Realtime Response Under Load Best Effort (RRUL_BE)

ICMP latency in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Download bandwidth, upload bandwidth and ICMP latency in box graph:

TCP download

Ping latency in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Download bandwidth and ping latency in box graph:

TCP upload

Ping latency in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) Upload bandwidth and ping latency in box graph:

8-streams download test

Ping latency in *ECDF* graph:

(**How to read:** closer to left is better, learn more about *how to read ECDF graphs*) 8 downloads bandwidth and ping latency in box graph: Article written by Toke Høiland-Jørgensen, Federico Capoano.

2.4 About duplicated packets

During the battlemesh v8 we uncovered a wifi driver bug which caused duplicated packets in certain cases.

Here's a few graphs that show the duplicated packets that were captured: Article written by Federico Capoano.

2.5 Reading (E)CDF graphs

An ECDF graph is very usefull to have a summary analysis of a big sample of very different values, but the first contact is quite surprising. Indeed, there is only one data represented on an ECDF graph, for example the RTT, while we are habituated to have one data in function of another, for example the RTT in function of the time of the experiment.

This document will explain you how to quickly interpret an ECDF graph: first with a simple example, then with a more general approach, and finally by showing you how to generate such graphs.

2.5.1 Quick example: reading a RTT CDF graph

In networking, the RTT (Round Trip Time) is the time taken by a packet to go from one computer to another, and to come back. It is common to measure it with the `ping` command. A CDF graph of the RTT of an experiment as the following will give you informations about the global performances of the network, as:

- almost all packets have more than 10ms and less than 50ms RTT,
- 80% of the packets are faster than 37ms,

but also about its fairness: here, we see that most of the packets are between 10 and 50ms. A perfectly fair network would looks like a vertical line (all packets have the same RTT).

2.5.2 Going deeper

A CDF (Cumulative **Distribution** Function) graph shows the **distribution** of the samples among values. Looking at CDFs can give you a quick view of pathologies, or specificities. The first example below shows you a segment going from one corner to the other one, on a wide range of values: the values are uniformally distributed. This is probably what you expect if, for example, you consider the timestamps of the collected samples. Having such result in an RTT graph would be more surprising.

The second example is more interesting: here there is two bearings at different values. The data are distributed in two sets around values 20 and 60. The y-axis gives you the amount of samples which have these values: here, we see that it's around 40% and 50% for the data having respectively 20 and 60 as values. Applied to RTT, it means that some packets travel faster than others, with a clear separation: there should be a good reason, but of course, the graph will not tell you which!

2.5.3 Understanding and drawing ECDF

As stated bedore, an ECDF represents the distribution of the samples among their values: the order in which data are provided doesn't matter, but only the values of the data. Let's consider the following sample collection, with heigh values.

Sample number	1	2	3	4	5	6	7	8
Sample value	3	2	2	3	1	2	5	2

A simple distribution representation would fit in one dimension, with plots, or vertical bars, dispached on an axis representing the values of the samples.

Sample value (sorted)	1	2	2	2	2	3	3	5

Of course, this kind of representation didn't hide the fact that some samples have the same value. The natural idea here would be to use some histogram-like representation, with vertical bars which the number of the samples in y-axis.

Sample value (sorted and aggregated)	1	2	3	5
Count	1	4	2	1

This will work for this example, but becomes quickly useless when we have many values, near but slighly different. To solve this issue, the ECDF is just the same, with cumulated number of samples and step plotting. The cumulated number of samples is then rebased on 1, which gives the probablility of the distribution (which is think by common mortals as percentage).

Sample value (sorted and aggregated)	1	2	3	5
Count (cumulated)	1	5	7	8
Count (cumulated and rebased)	1/8	5/8	7/8	8/8

Another and equivalent way to compute ECDF is to directly associate the cummulated numbers to the original (sorted) samples. Indeed, we remark that each sample contribute to 1/(number of samples) in the y-axis.

Sample value (sorted)	1	2	2	2	2	3	3	5
Cumulated sample number	1/8	2/8	3/8	4/8	5/8	6/8	7/8	8/8

2.5.4 Important remark

When you are performing an experiment, there may be trailing data which are not really part of the experiment. In a classic graph, it doesn't really matter, because we naturally see that when the curve is going to be flat and without any interest, then the experiment should be on its end, and we naturally don't pay much attention to such singularities. However, in ECDF graph, results are aggregated, and there is no way to know from where comes the packets. It is therefore important to take care of removing trailing data before processing an ECDF. The following graph illustrate this issue: we see that the first bearing should not be represented in the ECDF graph.

Article written by Matthieu Boutier.

2.6 Lessons learned

Lessons learned during the battlemesh v8.

2.6.1 Firmware

Problems:

Choosing to compile the firmware using openwrt trunk in the first days of the event was a mistake which delayed the testing phase.

Not stating clearly which protocols were going to participate in the battle caused misunderstandings and discussions which could have been avoided with more preparation.

Possible solutions:

- use a stable OpenWRT release
- choose a specific revision at least one month in advance
- prepare the firmware before the event

- the protocols that want to participate in the battle have to state it clearly in advance and have to take part during the test process

2.6.2 Configuration

Problems:

Planning the address scheme, configuring all the protocols to work together was quite a big deal of work.

We assumed the configurations from the previous years could be reused, but they had to be reworked from scratch.

Possible solutions:

- prepare a working configuration before the event
- ask all the routing protocol developers to contribute to the proposed configuration

2.6.3 Communication

Problems:

During the entire process communication between teams was uneffective and disorganized, a second testbed was set up manually, there were rumors circulating and this caused distress and loss of motivation of some people.

Possible solutions:

- have short meetings each day to debrief on the status of the test process and talk about eventual problems and proposed solutions
- take notes of the meetings and make them available to all the participants (eg: etherpad)
- use a projector and a microphone to keep attention levels high

2.6.4 Volunteers

Problems:

There were quite some people that wanted to help out but it was not clear how they could help because the tasks list contained only **firmware preparation** and **test management**, which were not enough to describe the entire test process to new comers so they were not able to understand how they could contribute.

Possible solutions:

- **encourage people to participate to one or more of these following teams:**
  - **planning team**: plans realistic test scenarios, that is, according to the situation of each event (number volunteers, location and so on)
  - **firmware team**: prepares the firmware with all the required agreed packages
  - **flashing team**: flashes devices in mass
  - **configuration team**: reviews config of the past year, checks if they can be reused as is or need to be modified, if modified gets them approved by the routing devs
  - **deployment team**: deploys the testbed and works until everything works correctly, updating configs if necessary
  - **test team**: writes and tests the scripts to execute tests and passes the raw data to the graph generation

  - **graph generation team**: writes (or reuses an already written set of) scripts to generate graphs from raw data extracted from the test scripts
  - **documentation team**: prepares drawing of topology, takes photos of the testbed for the presentation, writes an outline of the test plan, stores all the configs and scripts used during the process for later publication
  - **routing team**: developers of all the routing protocol involved oversee the whole process, with particular attention to the test plan phase, configuration and test scripts
- have people work in parallel on different indipendent tasks to speed up the process

Article written by Federico Capoano.

2.7 Credits

Many thanks to all the contributors who actively participated in producing and documenting the results of this edition.

2.7.1 Management and testbed setup

Management and test planning:

- Amadeus Alfa
- Juliusz Chroboczek

Testbed setup and routers flashing:

- Amadeus Alfa
- Federico Capoano
- Goran Mahovlić
- Tomislava
- Dario Stelitano
- Riccardo Bloise

2.7.2 Test and graph scripting

- Henning Rogge (test scripts)
- Thijs Van Veen (test scripts)
- Matthieu Boutier (graph scripts)

(*The mesh of death adversity*, *The crossed streams jeopardy*)

2.7.3 Flent tests

- Toke Høiland-Jørgensen

(*Blowing up the network test*)

2.7.4 Web site and documentation

- Federico Capoano
- Matthieu Boutier
- Thijs Van Veen

2.7.5 All contributions (with more details)

may be incomplete, feel free to send pull requests with additional names.

- Amadeus Alfa: test management and overview
- Antonio Quartulli: configuration and debugging
- Axel Neumann: configuration
- Dario Stelitano: flashing and configuration
- Dave Täht: configuration and testing of minstrel-variance patches
- Federico Capoano: flashing, configuration and documentation
- Goran Mahovlić: flashing and configuration
- Henning Rogge: configuration, debugging and test scripts
- Juliusz Chroboczek: configuration and test planning
- Matthieu Boutier: data analysis and graph generation
- Marek Lindner: configuration and debugging
- Riccardo Bloise: flashing, configuration and debugging
- Simon Wunderlich: configuration and debugging
- Sven Eckelmann: debugging wifi driver bug
- Thijs Van Veen: test scripts
- Toke Høiland-Jørgensen: *Blowing up the network test* (test, scripts and graphs)

Article written by Federico Capoano, Matthieu Boutier.

Article written by Federico Capoano, Matthieu Boutier.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search